CUDA method for the FDTD simulation by GPU

Wonhak Son

Department of Natural Science

Graduate school of UNIST

CUDA method for the FDTD simulation by GPU

A thesis

submitted to the Graduate School of UNIST

in partial fulfillment of the

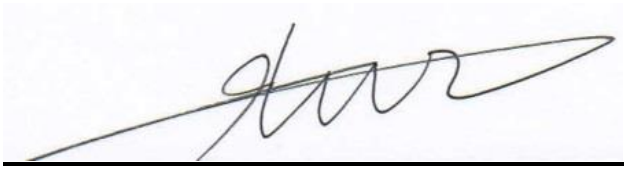
requirements for the degree of

Master of Science

Wonhak Son

6. 11. 2014 of submission

Approved by



Advisor

Min Sup Hur

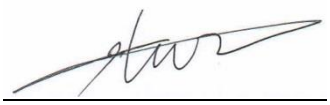
CUDA method for the FDTD simulation by GPU

Wonhak Son

This certifies that the thesis of Wonhak Son is approved.

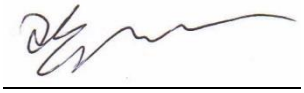
6. 11. 2014 of submission

Signature



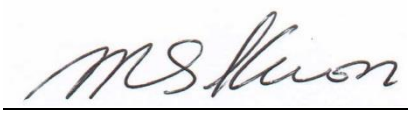
Advisor: Min Sup Hur

Signature



Typed name: Kyujin Kwak

Signature



Typed name: Min-Suk Kwon

Abstract

The technology of computational devices has been developed over several decades especially graphic processors which not only deal with graphic works but also compute scientific problems. This processor is suitable for parallel computations instead of using expensive high-end devices. Many research groups have implemented parallel computations using the MPI method with multi CPUs to solve difficult and complex problems. Lately, the CPU clock speed, due to technological limitations. Thus, the MPI method could not be speed up by adding CPUs. The GPU consisting of many threads is one solution for the technological limitation and can make the computation faster. Recently, a CUDA development environment has been developed by NVIDIA. It makes parallel computations easier for researchers dealing with complex problems. The FDTD simulation, a useful computational electromagnetic method was chosen for the parallel computation. In this paper the comparison between the MPI method and the CUDA method is discussed, then the FDTD simulation is tested using the MPI method and the CUDA method. Additionally, a simulation of particle movement in the FDTD simulation is implemented. This experiment enables the PIC simulation which needs a great deal of time due to the large number of particles. Since the GPU has a lot of threads to deal with these particles, it is possible to downsize the simulation time using the CUDA method.

Contents

List of figures

List of tables

1. Introduction

Recently many parallel programming skills are used in science to calculate difficult and complex problems. These contain a great amount of data which could not be treated in person. Although computer science has developed rapidly for a few decades and distributed in the public domain, there are still many problems which take a great time to solve or are impossible to compute correctly. Some computers, which are very expensive and difficult to treat, need much time to achieve their goal. In order to settle these difficulties, developers have looked for novel ideas and they have tried to find alternative methods for parallel calculations. A new method known as MPI, which stands for message processing interface, has been introduced for parallel calculation to reduce the computation time and efforts needed to solve complex problems. This method can also save memory by distributing data to each node computers and decrease the expense to make the computing system. The MPI method is an invaluable solution since the speed of the CPU has faced technological limitations. This method, however, cannot use all the computational simulations since the deadlock will happen as with using the dependable variables in the parallel computation. Therefore, it is important that the simulation using the parallel computation should be checked first. Then, the efficiency of the parallel simulation should be considered compared with the result from a single PC. Sometimes, parallel calculations need to transfer a lot of data between them in order to calculate. This work takes up a lot of time which is longer than the runtime using a single PC.

The MPI method using multi CPUs has a higher performance ability and is faster than single a CPU. This can be very useful, even though, the efficiency of the MPI method is not always good. In the FDTD (Finite Difference Time Domain) simulation that consists of a number of cells, the electric field data in a cell needs magnetic field data in the neighbored cells and vice versa. In the simulation, all data in the cells have to communicate with one another to implement the experiment. Thus, the MPI method for the FDTD simulation does not give good efficiency. For this reason, a novel technique CUDA (Compute Unified Device Architecture) method was introduced to the computational research group [1]. The GPU is originally devised for the graphic process to be distinct from the CPU calculating arithmetic problems in the parallel computation. These days the graphic process needs a high performance GPU which can deal with two-dimensional work as well as a three-dimensional process [2]. Thus, in order to meet the demands of the conditions, high-end graphic cards were developed. The development environment for using GPU was too difficult to treat at the beginning, the introduction of the CUDA development environment make it much easier.

Fundamentally, the speed of the GPU processor is slower than the CPU processor. However, since it has a number of threads which can compute in the parallel simulation, this method is very useful. While the MPI method using multi CPUs lets each CPU calculate the work that is divided by the total number of CPUs, the CUDA method is when all threads calculate their work divided by the thread number [8]. All threads work only one time in the simulation. In this paper, Yee Mesh code was introduced for the FDTD simulation and two methods were implemented and their results compared. The FDTD simulation is good for parallel computation because the domain consists of many discrete cells that can be divided by the number of CPUs or threads. The electromagnetic computation based on the Maxwell's Equation was dealt with by the FDTD simulation to solve complex and difficult problems that are hard to undertake in the real world. The new skill, CUDA method, enables many researches to complete their tasks.

The CPU, Intel Core i7-4770 3.40GHz, and graphic card, NVIDIA GEFORCE GTX Titan Black, were used in the simulation, the Operating System was 64bit Linux CentOS6.4. The graphic card, GEFORCE GTX Titan Black has a special feature that is a 384-bit GDDR5 memory, bandwidth 336GB/sec, and 6GB memory. In order to display the outputs from the simulation, VisIt software from Lawrence Livermore National Laboratory (LLNL) was used. This application enables us to display the two-dimensional picture as well as a three-dimensional object. In addition it can make animations from the results, so it is possible to watch the motion of beam and particles. It needs two formats which are the hdf5 format and the silo format. The hdf5 format, developed in the National Center for Supercomputing Applications, was used to store electromagnetic field data. The silo format devised in the LLNL was used to store particle information. These were used the following sections.

2. FDTD theory

The finite difference time domain (FDTD) method is one the most efficient computational techniques used to approach Maxwell Equations which are used in various field of science for the analysis of electromagnetic phenomena, sound, and heat transfer [8]. This technique makes it possible to approach many difficult experiments in person. However, this simulation has a few limitations because it needs a large amount of time to achieve the correct results.

Fundamentally, the FDTD method based on the Maxwell Equations compute electromagnetic fields in time and space. It consists of one, two, and three dimensional equations that are not very different. Therefore, in this paper, two and three dimensional computational method are presented using the CUDA algorithm.[6]

$$\frac{\partial \vec{B}}{\partial t} + \nabla \times \vec{E} = 0 \tag{1}$$

$$\frac{\partial \vec{D}}{\partial t} - \nabla \times \vec{H} = \vec{J} \tag{2}$$

$$\vec{B} = \mu \vec{H} \tag{3}$$

$$\vec{D} = \varepsilon \vec{E} \tag{4}$$

From the these equation each electric field and magnetic field can be presented

$$\frac{-\partial \vec{B}_x}{\partial t} = \frac{\partial \vec{E}_z}{\partial y} - \frac{\partial \vec{E}_y}{\partial z} \tag{5}$$

$$\frac{-\partial \vec{B}_y}{\partial t} = \frac{\partial \vec{E}_x}{\partial z} - \frac{\partial \vec{E}_z}{\partial x} \tag{6}$$

$$\frac{\partial \vec{B}_z}{\partial t} = \frac{\partial \vec{E}_x}{\partial y} - \frac{\partial \vec{E}_y}{\partial x} \tag{7}$$

$$\frac{\partial \vec{D}_x}{\partial t} = \frac{\partial \vec{H}_z}{\partial y} - \frac{\partial \vec{H}_y}{\partial z} - \vec{J}_x \tag{8}$$

$$\frac{\partial \vec{D}_y}{\partial t} = \frac{\partial \vec{H}_x}{\partial z} - \frac{\partial \vec{H}_z}{\partial x} - \vec{J}_y \tag{9}$$

$$\frac{\partial \vec{D}_z}{\partial t} = \frac{\partial \vec{H}_y}{\partial x} - \frac{\partial \vec{H}_x}{\partial y} - \vec{J}_z \qquad (10)$$

From these equations, you might think that there is no current because the only source is an incident wave. The boundaries of the domain could be a perfect conductor and the size of each cell should follow the equation (11).

$$\sqrt{(\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2} > c\Delta t = \sqrt{\frac{1}{\varepsilon\mu}}\Delta t \qquad (11)$$

The distribution of the electromagnetic field data in the FDTD simulation is shown in Fig.1

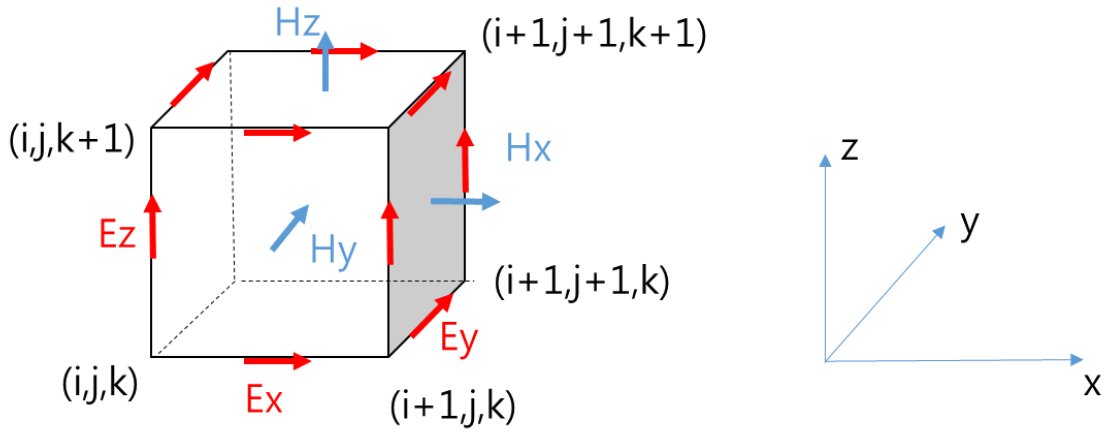


Fig. 1. Position and direction of each field component in the three-dimensional domain.

The position of the electric field and the magnetic field are in the domain located side and face. However, it is not important where they were but that all components would allow each point to be crossed by one another. This schematization helps to understand the position of each component data in the FDTD simulation. A domain in the FDTD simulation consists of cells whose sides are dx, dy, dz. All cells have to store their information including electric, magnetic field data, and particles. A cells' array in the memory should be secured to compute the simulation. It is so important what the exact cell's number is including the position of the electric and magnetic field in the domain. In the FDTD

simulation, information of the neighbored cell's position is required to obtain information from them to calculate electric and magnetic fields in the current cell. The magnetic field could be induced by closed the electric field and the electric field could be created by an induced magnetic field according to the equations (5) ~ (10). To avoid deadlock, each field data should be taken in turn to compute every time. In this case, an error would occur since the electric field and magnetic field are induced simultaneously. Thus, setting a short time step in the simulation could reduce the error. In this simulation, time step Δt was set at $8.339 \times 10^{-17} s$. The flowchart of the FDTD simulation is shown in Fig. 2

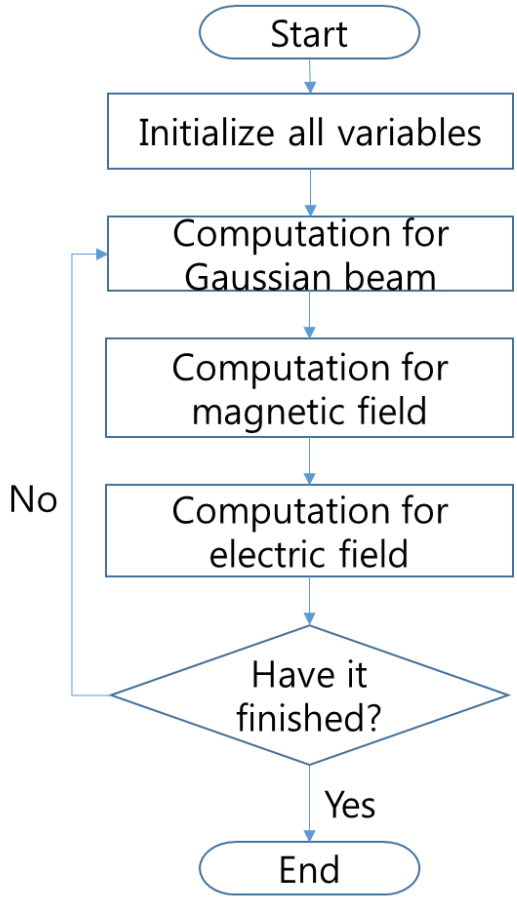


Fig. 2. A flowchart of the FDTD simulation

In the Fig. 2 the main loop consists of three parts. The incident wave, and the Gaussian beam, should be processed after initializing all variables. Then, the procedure for computing the electric and magnetic fields was implemented by taking turns. In this paper, computation was implemented 10,000 times.

3. Two-dimensional FDTD simulation

Although the electromagnetic wave which is propagated in space is a phenomenon in three-dimensions, we may select a two-dimensional computation because the calculations of two dimensional electromagnetic wave equations are similar to three dimensional equations. In this case we do not need to use all the equations from (5) to (10) but can select a few equations that we need. In this paper, we set the source of FDTD as a Gaussian beam which is polarized in the y direction and propagated the x direction. In this case we need three equations for two electric fields in the x and y direction and one magnetic field in the z direction. The size of whole domain consists of a width of $3 \times 10^{-4}m$ and a height of $3 \times 10^{-4}m$ and it has a dx of $5 \times 10^{-8}m$, and a dy of 4×10^{-7}m. In order to get the number of the total cells, the width must be divided by dx, and the height divided by dy which gives us $6{,}000 \times 750$ that is 4,5000,000. Since each cell has three set of data, namely two electric fields and one magnetic field, the cells have to save three fields of information which consist of a float type which has four bytes. Therefore, $6{,}000 \times 750 \times 3 \times 4 = 54\text{MB}$ of memory which is needed to simulate this two-dimensional FDTD calculation. The following figure presents the structure of a two dimensional FDTD and the direction of the electromagnetic field.

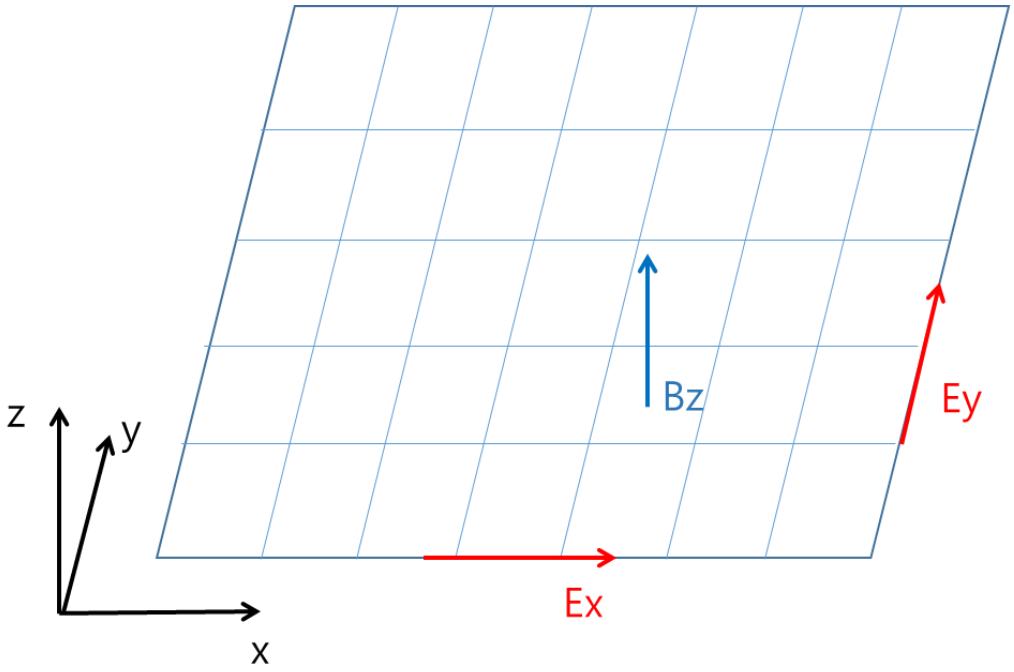


Fig. 3. Two electric field and one magnetic field in the two-dimensional FDTD simulation.

In order to compute the two-dimensional FDTD simulation we do not need all direction electromagnetic fields. It is not too different from the three-dimensional equations that are required for equations (5) through (9) but need (7), (8), and (9) so it can save time and memories. Next, each cell can compute the electromagnetic field according to these equations. There are two methods to compute the two-dimensional FDTD simulation, one is the MPI (Message Passing Interface) method and the other is the CUDA (Compute Unified Device Architect) method. These will be discussed in the subsequent sections.

3.1. MPI method

The MPI method, which uses many computers that calculate work divided by the total computers number, is similar to parallel computing method. Suppose there are one hundred cells in the domain and ten computers (i.e. ten CPU cores) to calculate the FDTD simulation, each computer only has to compute ten cells of problems. It can save time and memory because each computer only needs to calculate one tenth of the work. Theoretically this technique will be ten times faster than the technique with uses computer for the calculation. However, there are a few problems to overcome in this theorem. In the FDTD algorithm, neighbored cell's data is needed to compute the electromagnetic fields. Since the neighbored data which is separated from other computers should be transferred to the cell, it takes transferring time. Thus it is difficult to get the theoretical speed by adding more computers. Sometimes, data transferred via LAN (Local Network Area) need too much time because of the network traffic jam. That is, using many computers can reduce the calculation time and memory needed by sharing the work but it is difficult to reach the theoretical speed of simulation. Therefore users have to consider that the number of computers they will use to compute and the size of the data. The following experiment is a result of the calculation time from the different sizes of domain.

Three different domain sizes were set in the two-dimensional FDTD simulation to measure the runtime of the MPI method.

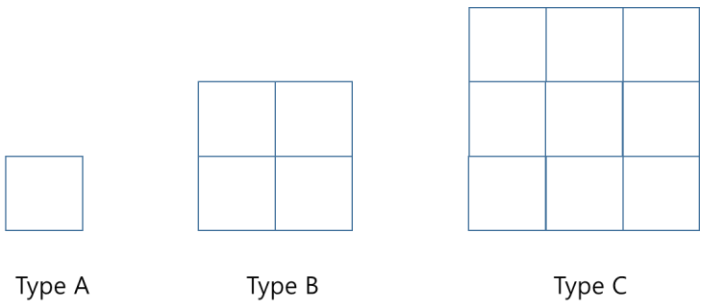


Fig. 4. There are three different domains to compare runtime in the two-dimensional FDTD simulation

Type A is a square domain which has a side of length $10^{-4}m$ and has $2000 \times 250 = 500{,}000$ cells in the area. Type B is a square domain which is four times bigger than the Type A domain and Type C is nine times bigger than Type A. Type B has 2,000,000 cells and Type C has 4,500,000 cells. When the simulation was implemented by one CPU type A took up a lot of time which is longer than others.

Area	CPU 1ea	CPU 2ea	CPU 4ea	CPU 8ea
type A	569.5	443.9	306.7	266.1
type B	2213	1549.3	1255.8	1061.7
type C	7365.8	3524.5	2814.3	2413.8

Table 1. Runtime from the different domain size as using different number of CPUs

When using only one CPU, type B took 2213 seconds that is four times longer that type A took to complete the simulation since the domain size of type B is four times the type A domain size. In the case of type C, it took more than nine times the runtime of type A. This is similar to the other case that the computation used by different number of CPUs. Generally, the FDTD simulation with large sizes of the domain need much time to complete the computation. The runtime is proportional to the size of the domain not the number of CPUs.

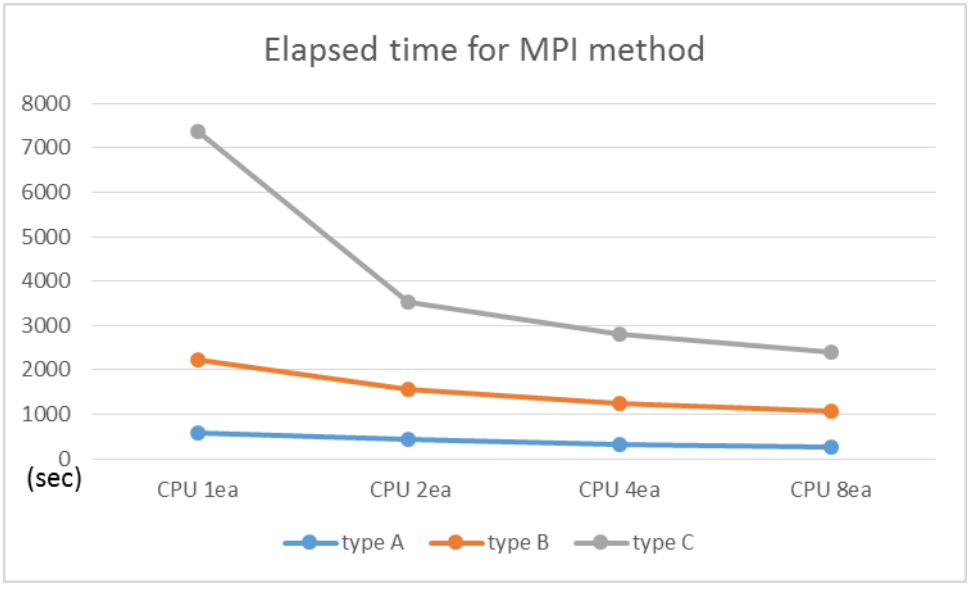


Fig. 5. Runtime of the simulation in MPI method

In general, the FDTD simulation using multiple CPUs is more efficient than a single CPU. However, researchers must consider the number of CPUs when computing the FDTD simulation using the MPI method. From Fig.5 the efficiency of the MPI method is not always the same. The MPI method with two CPUs is much more efficient than four CPUs or eight CPUs. This simulation was implemented with the Intel Core i7-4770 3.40GHz that has eight cores that are in the single computer. If one needs more CPUs to compute the simulation, another computer should be connected up via LAN. In this case, a bottleneck will occur on the network, for this reason it takes a great deal of time to compute the simulation and this has been reported in other experiments. Therefore, it is efficient when the two-dimensional FDTD simulation is implemented using the MPI method with huge amounts of data and decreasing the bottleneck as far as possible.

3.2. CUDA method

The CUDA method is very similar to the MPI method. In this section, almost of the source codes are based on the two-dimensional FDTD algorithm in the previous section. The CUDA method is implemented by the GPU core unlike the MPI method that is implemented by the CPU. While the MPI method is a computation in which each work is divided using the CPU's number is implemented, the CUDA method is implemented by whole threads that are made up of the whole work number, i.e. the total number of cells. The best feature of the CUDA method is that it does not need to use main loop state in the source code because all threads can do the work in one time. In the previous section, the two-dimensional FDTD simulation with eight CPUs on the calculation of type A, the number of calculations each CPU has to calculate was $500{,}000 \div 8 = 62{,}500$ i.e. one CPU must compute Maxwell Equations 62,500 times in the loop state. In the CUDA method, it is required to set 62,500 threads then each work of the cell could be designated to each thread. Thus all threads do 62,500 times the work at a time. As a matter of fact, the speed of the CPU is faster than the GPU in the system but the number of threads is bigger than the CPUs. This makes it possible for the runtime of the CUDA method to be decreased and become faster than the MPI method.

In using a graphic card which has small quantity of threads less than the total thread number 62,500 it is impossible to compute the two dimensional FDTD of type A one at a time. In this case, the domain should be cut up into small sections and the threads have to solve the each section in consecutive order. If there is not enough memory on the GPU board, whole data should be stored in the memory on the mainboard and be transferred from mainboard to the GPU memory and vice versa.

Transferring data requires much time for computation in the simulation it is very important to consider the amount of work and the number of total threads. In order to increase the efficiency, the amount of work should be less than the total number of threads, and memory size that is needed to compute on the graphic card must be able to hold the total data.

Another way to increase efficiency is to use the shared memory instead of using the global memory on the graphic card [3-5].

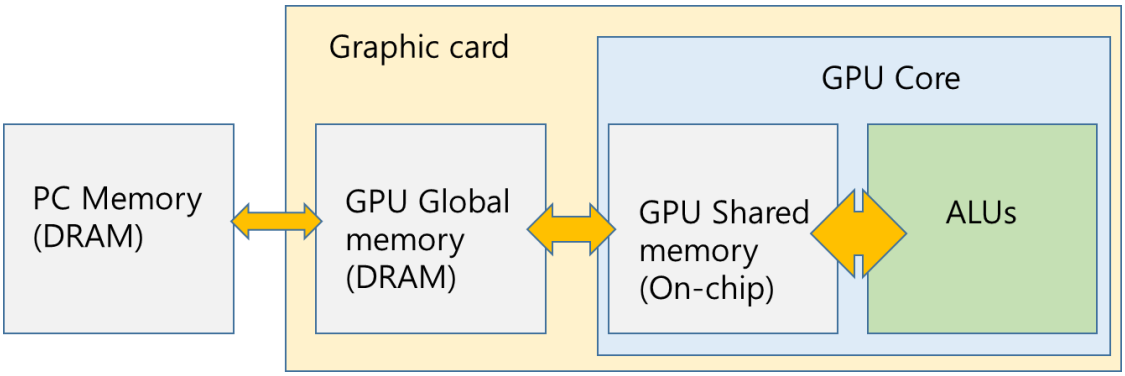


Fig. 6. Schematic of memory on the graphic card

All threads can access global memory and shared memory with the same lifetime as the block. Shared memory is an on-chip memory that are faster than global memory access to data. However, there is limitation concerning shared memory use because it has a small size. Thus, to get high efficiency the simulation it is a good method to reuse data in the shared memory.[20] In this paper, shared memory was not used. Although shared memory is faster than the global memory, it cannot include the whole data of the domain. Only divided data can be loaded to the shared memory and computed, so it is necessary to exchange data from the global memory to the shared memory many times. This also takes much time in the simulation, so we used only global memory.

CUDA method is appropriate for C and FORTRAN. In this paper, source codes were generated by using C and Fig. 7 presents threads hierarchy [12], [13].

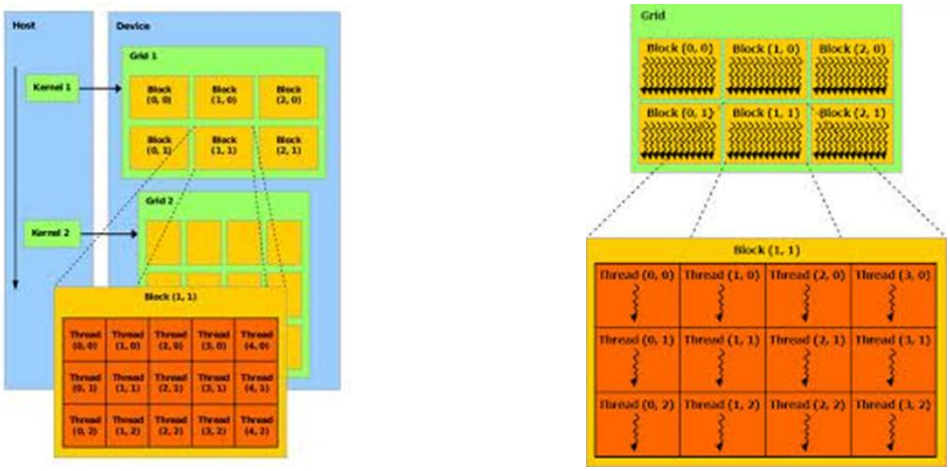


Fig. 7. Schematic of grid, block, thread in GPU. http://docs.nvidia.com/cuda/cuda-c-programming-guide

The grid consists of many blocks and each block is made up of threads. A thread calculating the electromagnetic field in a cell has its own ID that is accessible within the kernel through the built-in 'theadIdx' variable. One of the most important things in the CUDA implementation is to get the correct thread ID and allocate it to the proper work. Since all threads can access the global memory in the lifetime they can read and write data from the global memory.

A thread ID for x direction is included in built-in variable 'threadIdx.x. Another variable 'blockIdx.x' has an information of a position for x direction from the grid position. 'gridDim.x' and blockDim.x which are also built-in variables have the x-component number of the blocks and threads respectively. Likewise, y and z direction have the same components and variables. Thus, the unique thread ID can be gotten from a simple equation [7].

```
int x = threadIdx.x + blockDim.x*blockIdx.x;
int y = threadIdx.y + blockDim.y*blockIdx.y;
int id = x + gridDim.x*blockDim.x*y  + gridDim.x*gridDim.y*blockDim.x*blockDim.y*z;
```

Listing 1. This code resents a method to get thread unique ID in two-dimensional FDTD simulation

Suppose that there is a domain type C in the previous section which has 6,000 cells for the x direction and 750 cells for y direction. These cells needs memory to store their data, in the C source code, it is defined as two-dimensional arrays, array[6,000][750], for float type on the mainboard while it is defined as a one-dimensional array, array[$6{,}000 \times 750$], in the graphic card. Consequently,

it is necessary to calculate the correct array number for the CUDA method.

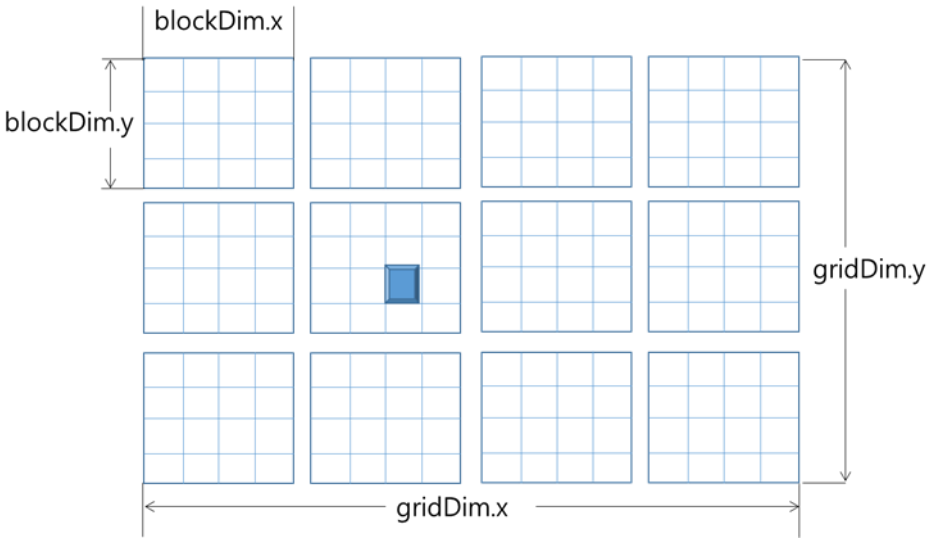


Fig. 8. A structure of thread, block, and grid

This Fig. 8 represents the position of the thread in the two-dimensional array. A grid consisting of a 4×3 array has a block with 4×4 array threads. That is, 'gridDim.x' is a four and 'gridDim.y ' is a three, and 'blockDim.x', 'blockDim.y' are four respectively. The thread ID marked position at Fig. 8 is a second row and second column of the block array and third row, third column in the block. Thus, thread ID is that $4 \times 4 \times 6 + 7 = 103$, however, the array number in C starts at zero, so correct thread ID is $4 \times 4 \times (7 - 1) + (7 - 1) = 102$, that is 102^{th} thread ID can be allocated to the 102^{th} cell. Fig. 8 shows that the domain consists of 16×12 cells and the marked array position is an array$[6][6]$ in the two-dimensional array. The 102^{th} thread can access to the array[102] and its neighbor array to compute the electromagnetic field.

In the two-dimensional FDTD simulation, in order to propagate the Gaussian beam in the domain, each cell must update its electric field and magnetic field each time. To update the electric field, neighbored magnetic fields are needed according to the Maxwell Equations (8), (9). The cell having 102^{th} thread ID needs the magnetic field data from the neighbor cells that are left (101^{th}), right (103^{th}), up (86^{th}), down (118^{th}). The magnetic field could be obtained from the Maxwell Equation (7) in the same way as the electric field. The Listing 2 shows that how to get the magnetic field in the two-dimensional FDTD simulation.

```
temp1 = x + gridDim.x*blockDim.x*(y+1);
temp2 = id + 1;
if((x+1 < xSize) && (y+1 < ySize))
gpu_Field[id].Bz+=((dt/dy)*(gpu_Field[temp1].Ex-gpu_Field[id].Ex)-(dt/dx)*(gpu_Field[temp2].Ey-gpu_Field[id].Ey));
```

Listing 2. CUDA code to compute magnetic field in the FDTD simulation

To get each magnetic field of the z-component data in the cell, it is required to get neighbored cell' electric field data for x-component and y-component. For example, computation of the magnetic field for the z-component needs neighbored electric field data Ex (i, j), Ex (i+1, j) and Ey (i, j), Ey (i, j+1). From the thread ID designated to the magnetic field Bz (i, j) neighbored cell's number could be calculated and saved. The variable temp1 in the Listing 2 saves the index of (i, j+1) and temp2 saves (i+1, j). When calculating the closed cell's number, it should be noted that these numbers must not exceed the boundary of the domain. If the number of temp1 or temp2 is out of range of the array number, the CUDA cannot execute the FDTD simulation any more. To get the electric field data the method is similar to the calculation for the magnetic field. Likewise, the variables temp1 and temp2 in the Listing 2 save closed cell's numbers. The temp1 saves index for (i, j-1) and temp2 saves for (i-1, j). The only difference is that the calculating position in the simulation for the electric field is one step behind the magnetic field. The Listing 3 presents the main source code of the two-dimensional FDTD simulation.

```
int main(int argc , char* argv[])
{
        dim3  block(200,50);
        dim3  thread(30,15);
        float time = 0;
        int   count = 0;

        Init();

        updateBoundary<<<block,thread>>>(gpu_Field,time,dy,yDomainSize,yMesh);
        while(count <= TOTAL_CALCULATION)
        {
                BProc<<<block,thread>>>(gpu_Field,dt,dx,dy,xMesh,yMesh);

                if(!(count % SAVE_INTERVAL))
                        saveField_hdf5(count);

                EProc<<<block,thread>>>(gpu_Field,dt,dx,dy,xMesh,yMesh);
                updateBoundary<<<block,thread>>>(gpu_Field,time,dy,yDomainSize,yMesh);
                count++;
                time += dt;
        }
        Finalize();
        return 0;
}
```

Listing 3. Main source routine of the FDTD simulation

There is a peculiar symbol, that is ’ <<< >>> ‘ , it means that a function using in the programming is not controlled by the CPU but by the GPU. All variables in the GPU can be used in the same way as variables in the CPU. All data can be transferred between the CPU and the GPU but the address value of the GPU and the CPU cannot send or receive each other. The CPU cannot access the any of the memory on the graph card and the GPU also cannot access the mainboard. Thus, the same data for the FDTD should be created and stored in both memories. When computing the electromagnetic field using the CUDA method, the data in the graphic card is needed and when saving data to the storage device, calculated data should be transferred to the memory on the mainboard then saved. Fig. 9 represents elapsed time of the two-dimensional FDTD simulation using the CUDA method.

	type A	type B	type C
runtime	5.222	15.312	34.556

Table 2. Runtime by using CUDA method with different size of the domain

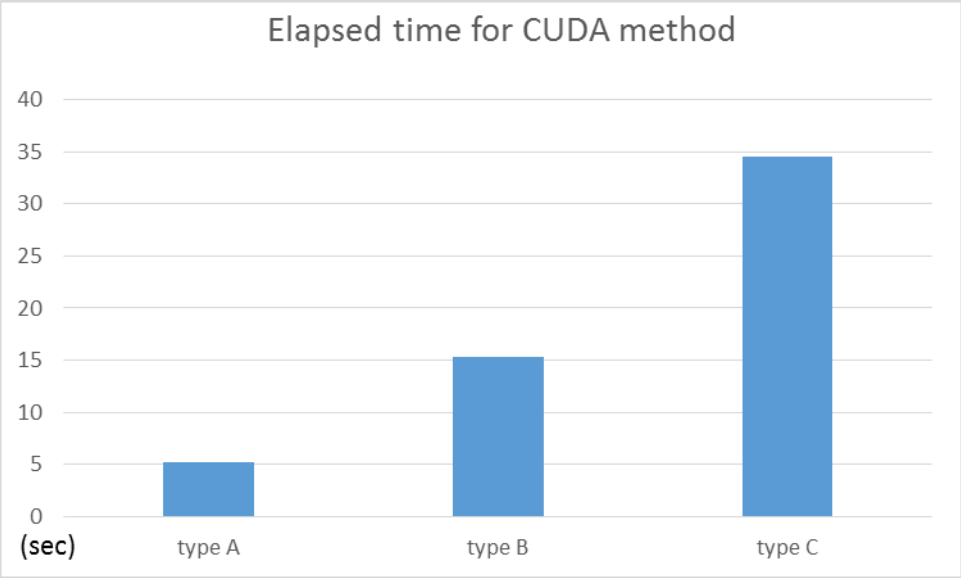
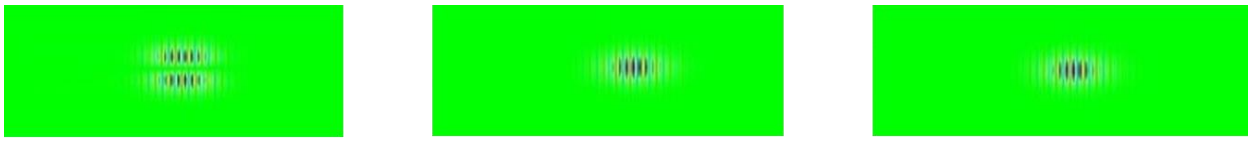


Fig. 9. Runtime of the simulation in the CUDA method

Compared to the MPI method, the CUDA method is faster and very valuable for the FDTD simulation. Computation for type A by the CUDA method took about 5 seconds. In the MPI method, the least runtime for type A took 266 seconds using eight CPUs. It means that computation by the CUDA method is more than fifty times faster than the MPI method.

3.3. Results

From the computation for the two dimensional FDTD simulation, the outputs could be displayed by graphic tools. Fig. 10 shows the electromagnetic field.



Ex field Ey field Bz field

Fig. 10. Each component of the Gaussian wave in the two-dimensional was displayed by using VisIt software

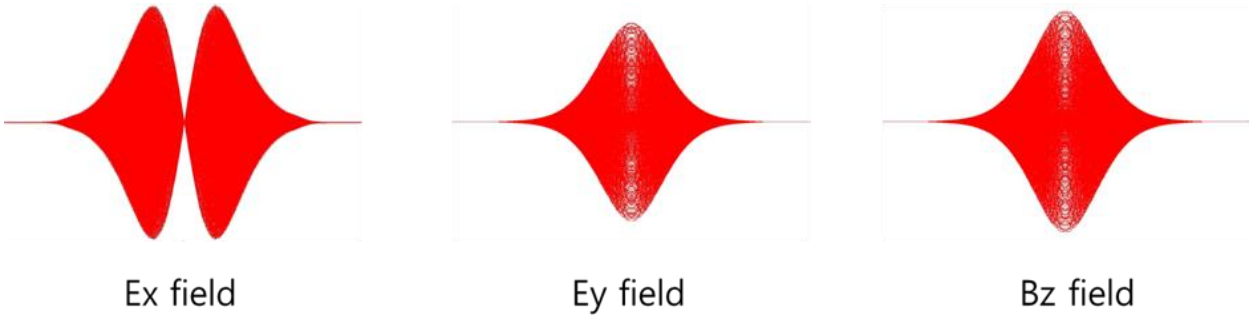


Fig. 11. Each picture showed cross sectional field intensity from the Fig. 10

4. Three-dimensional FDTD simulation

Now let us turn to the three-dimensional FDTD simulation using the CUDA method. In this section, we do not deal with the MPI method since that method needs a great deal of time to complete and is not much different to the two-dimensional FDTD simulation. The three-dimensional FDTD simulation needs all Maxwell's Equation (5) ~ (10) to obtain all directions of electromagnetic field data. In the three-dimensional simulation, a large amount of memory is needed to store all data, thus the range of memory size and thread number must not be exceeded. As a matter of fact, the CUDA method is not valid for massive parallel programming including hundreds of gigabytes since the graphic card has limited memory size and threads. Since the graphic card, GEFORCE GTX Titan Black, has a 6GB memory used this simulation, the data for the domain must have a smaller data size than 6GB. There is a three-dimensional cubic domain that has one side of length $10^{-4}m$ with 2,000 cells in the x direction, 250 cells in the y direction, and 250 cells in the z direction. The number of total cells is $2{,}000 \times 250 \times 250 = 125{,}000{,}000$ and each field data was defined as float type, i.e. four byte. Since three directions for electric field and magnetic field are needed, the total memory size of $125{,}000{,}000 \times 6 \times 4 = 3{,}000{,}000{,}000$ bytes = 3GB is required to implement the simulation. Fig. 12 shows a structure of the three-dimensional thread.

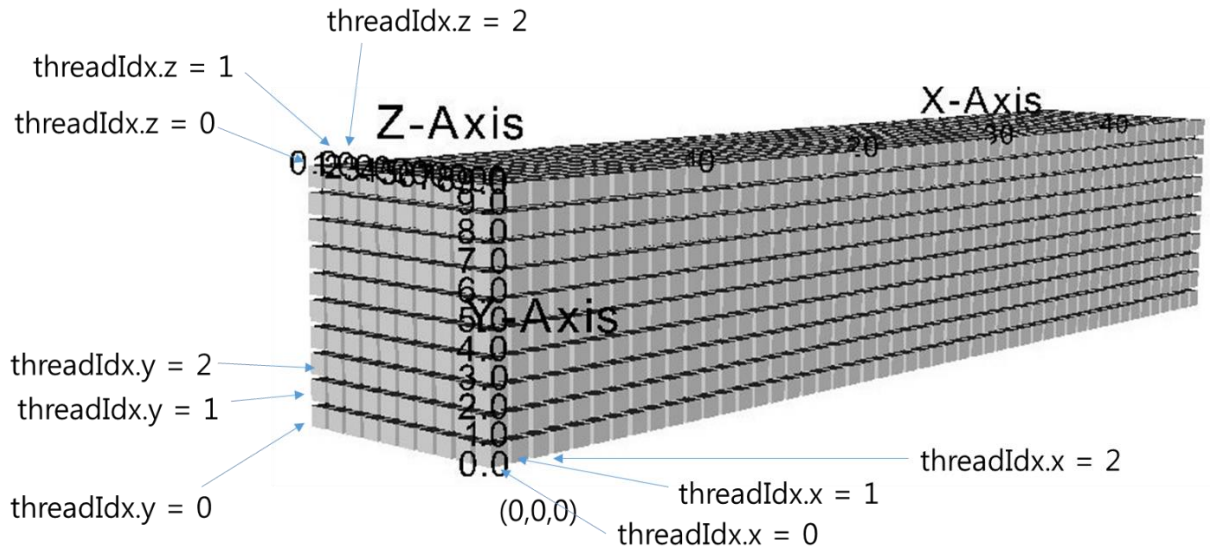


Fig. 12. A structure of the thread in three-dimensional FDTD simulation

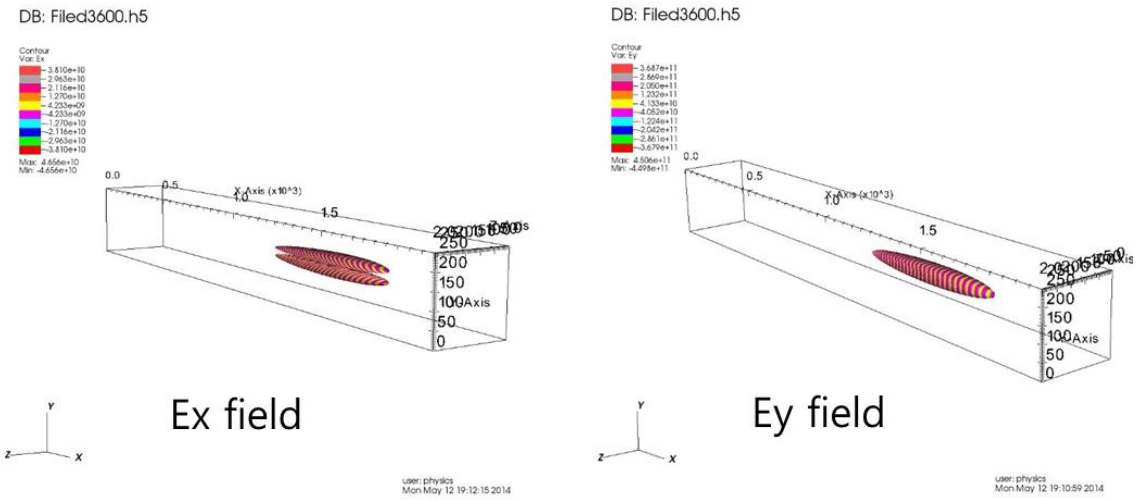
Likewise with the two-dimensional FDTD simulation, unique thread ID should be obtained in the three-dimensional FDTD simulation too. The way of getting the correct ID is similar to the case of the

two-dimensional simulation. Unlike the two-dimensional simulation, the z axis was added to the three-dimensional simulation.

```
int x = threadIdx.x + blockDim.x*blockIdx.x;
int y = threadIdx.y + blockDim.y*blockIdx.y;
int z = threadIdx.z + blockDim.z*blockIdx.z;
int id = x + gridDim.x*blockDim.x*y  + gridDim.x*gridDim.y*blockDim.x*blockDim.y*z;
```

Listing 4. Thread ID in three-dimensional FDTD simulation

Listing 4 represents that how to get thread ID from the each directional component. With the z-component in Listing 4, the source code is similar to getting the two-dimensional thread ID in the two-dimensional simulation. Fig. 13 shows the Gaussian beam propagating in the three-dimensional domain was displayed using the VisIt tool.



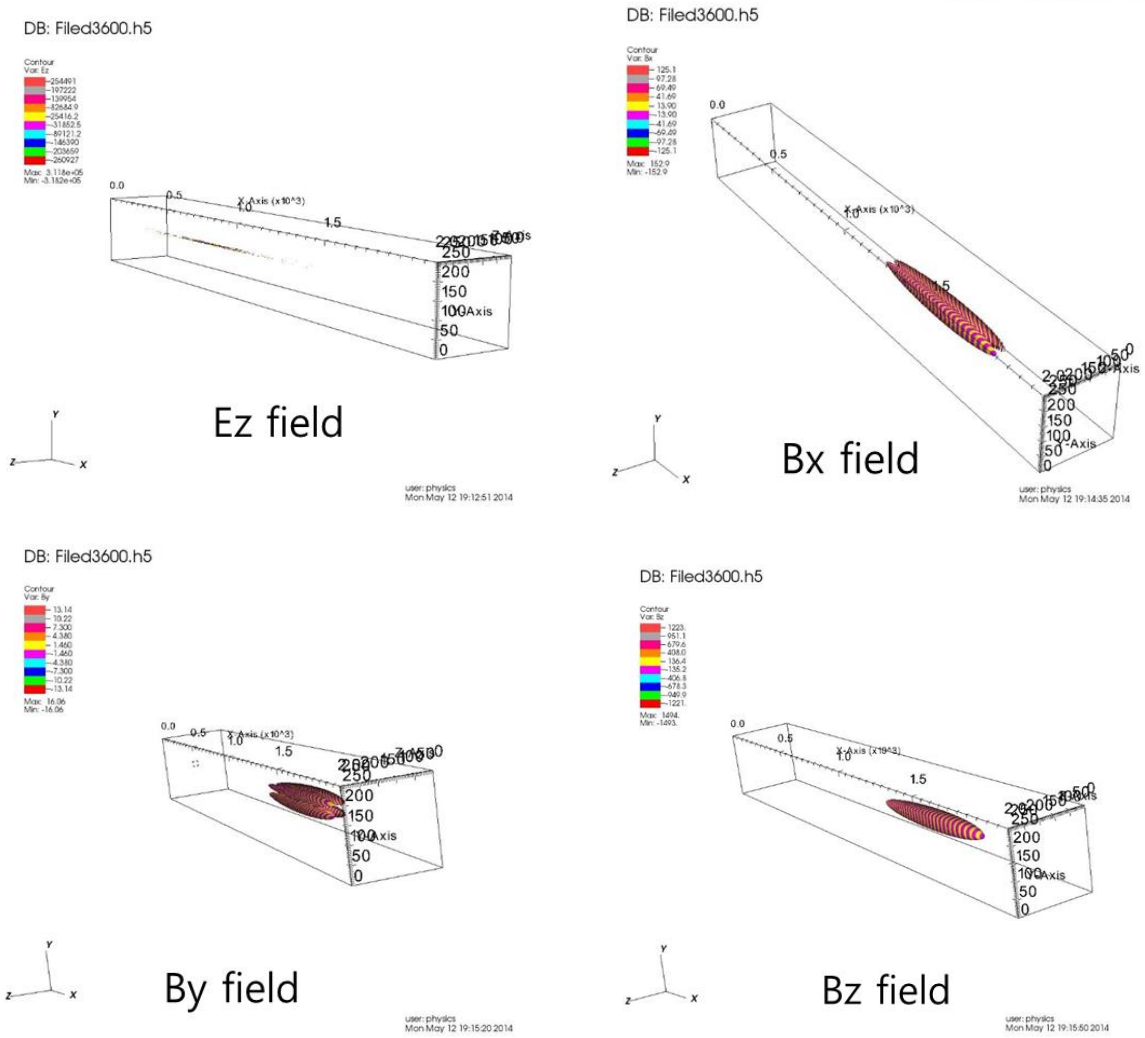


Fig. 13. Propagating Gaussian beam in three-dimensional FDTD simulation

5. Particle movement

In this section, we discuss the particle movement in the FDTD simulation using the CUDA method. A particle having mass and electric charge could be moving by the Lorentz force in the electromagnetic field. This phenomenon has contributed much to our life in various areas of study and science. Lately, many researchers have studied an accelerator to prove scientific theory or find new material in the world. This attempt, however, was very difficult and needed expensive facilities. The particle movement in the FDTD simulation is one of the best solutions for this difficulty. In this section, we discuss the particle movement in the FDTD simulation by the CUDA method. Since the simulation with particles usually set a number of particles that may exceed the range of the memory and threads, one should be careful as before. The thread ID which will be allocated to each particle does not need to be calculated since the particles do not follow the Maxwell's Equation but Lorentz force. When a particle is created in the simulation, the thread ID can be allocated to it one by one. If there were one hundred particles in the simulation, the thread ID could be set in order from 0 to 99. That is, a thread with ID zero calculates the first particle, the next thread calculates the second particle and so on.

Some information about a particle can be stored to compute the movement in the simulation, Listing 5 shows the structure of the particle information in two-dimensional simulation.

```
typedef struct PARTICLE{
        float prev_x;
        float prev_y;
        float  x;
        float  y;
        float  ux;
        float  uy;
}PARTICLE;
```

Listing 5. A particle structure

These values should be set at the beginning of the simulation. Variables x, y are the particle's position and ux, uy are relativistic velocity and prev_x, prev_y are the previous position of the particle. After setting all particles' information on the mainboard, this data should be transferred to the global memory on the graphic card. Then, each particle in the domain needs the correct cell position where it is now to get its electromagnetic data in the cell to compute the particle movement.

```
xPos = (int)((ptcl[id].x/dx)  + 0.1);
yPos = (int)((ptcl[id].y/dy)  + 0.1);
index = yPos*xMesh + xPos;
```

Listing 6. xPos and yPos mean normalized position for x direction, y direction respectively. The index in code means array number.

xPos and yPos in Listing 6 are array indices in the two-dimensional array. In the global memory, however, arrays consists of a one dimensional array, so an index for a one dimensional array should be converted by using the variables xPos and yPos. The electromagnetic field in the cell causes the particle to move according to the Lorentz force that are these equations [9-11].

$$\vec{F} = m\vec{a} = q\left(\vec{E} + \vec{u} \times \vec{B}\right) \quad (12)$$

$$\vec{a} = \frac{q}{m}\left(\vec{E} + \vec{u} \times \vec{B}\right) \quad (13)$$

$$\vec{u} = \vec{u}_0 + \vec{a}t \quad (14)$$

$$\mathrm{s} = s_0 + \vec{u}_0\mathrm{t} + \frac{1}{2}\vec{a}t^2 \quad (15)$$

There is one problem in using these equations in the FDTD simulation due to the leapfrog algorithm. The electric field and magnetic field are generated simultaneously in the real world. The electric field induces the magnetic field and vice versa. In this case, the electric and magnetic field data in the equations (12), (13) cannot be used because both field data were not created at the same time. These equations, however, we can use because the computation for the time step which is 0.8339×10^{-16} sec does not have a big error. Thus, the electric and magnetic field data in the cell can be used for the equation (12), (13) to get a particle's acceleration.

Another problem that will occur as these equations are used without considering the special theory of relativity. A particle, in this section we used electron, has very small mass, so the acceleration might have very huge value in the electromagnetic field. The velocity in equation (14) may exceed the speed of light for a time step. The relativistic velocity $\vec{u}$ was used instead of normal velocity $\vec{v}$. Listing 7 presents these equations.

```
beta=(ux*ux+uy*uy)/(c*c);
gamma=1/sqrt(1.0-beta);
ax = Ex*q/m + Bz*uy*q/m;
ay = Ey*q/m – Bz*ux*q/m;
ux += (ax*dt/gamma);
uy += (ay*dt/gamma);
beta=(ux*ux+uy*uy)/(c*c);
gamma=1/sqrt(1.0-beta);
x += (ux*dt/2/gamma);
y += (uy*dt/2/gamma);
```

Listing 7. Computation of the particle movement

Every particles needs their previous position and velocity to calculate the new position and velocity. In the FDTD simulation, the domain consisted of a number of cells that contain their own electromagnetic field data which affect particle's movement and direction. All particles are required to get the position where they are and the cell which they are put in. Then, the particles could be calculated using the electromagnetic field from the cell where they are located now. The gamma factor in Listing 7 adjusts the velocity and must not exceed the speed of light. Fig. 14 shows particles' movement in the two-dimensional simulation. As the Gaussian beam is propagating, particles were affected by the beam's electromagnetic field in the cells and moved. All particles' information were saved every 200 times, each picture was from the VisIt tool.

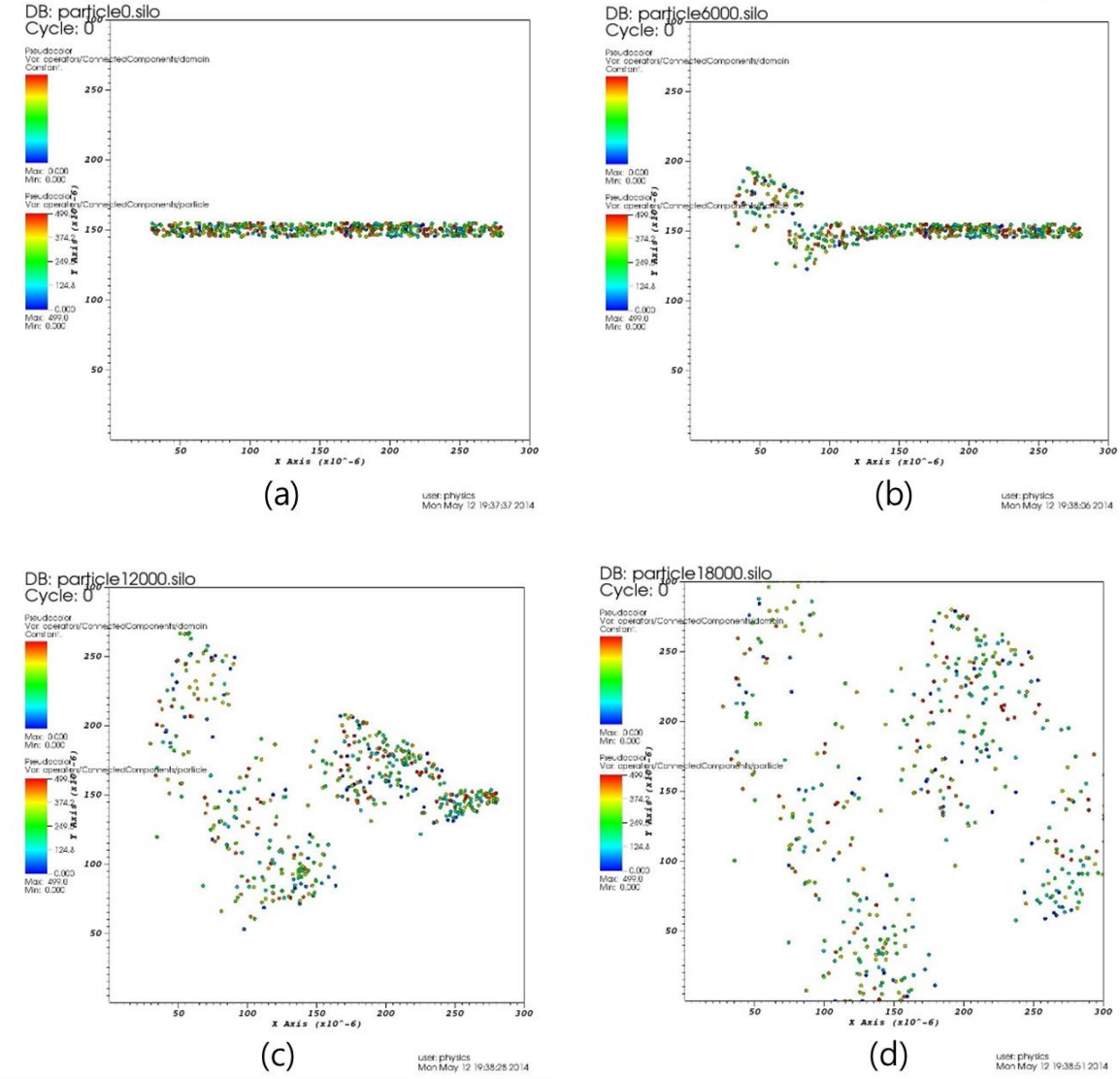
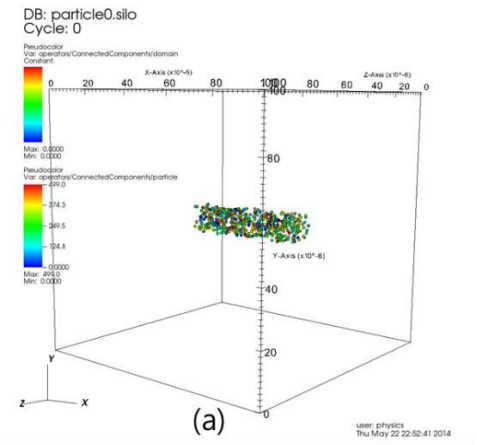


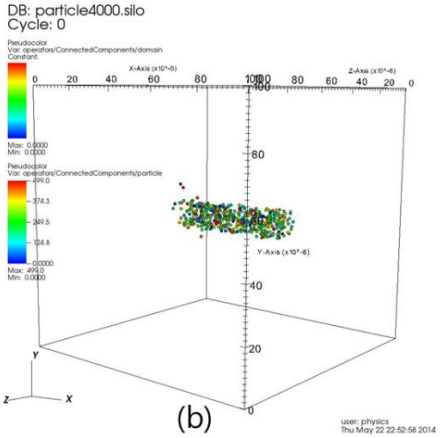
Fig. 14. Particles movement by Gaussian beam in the two-dimensional FDTD simulation

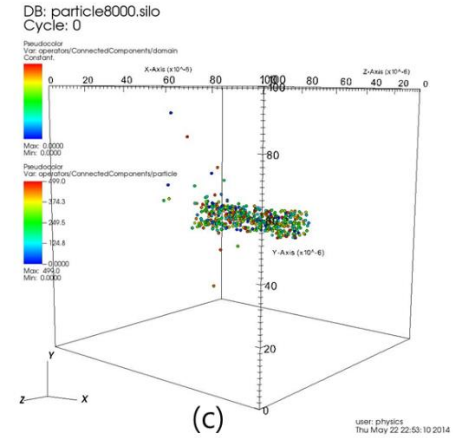
The Gaussian beam's focus was set at $3 \times 10^{-5} m$ where the left end particle's position, so the intensity of the beam was the strongest here. All particles were located randomly in the domain and formed a band to be affected by electromagnetic field easily. The source polarized in the y direction the Gaussian beam has the strongest in the y direction electric field than the other direction field in the beginning which affects particle to further move in the y direction. Thus, most particles moved up and down much more than left or right. In the simulation, the amplitude adjusted to 1.2×10^{12} to see the particle movement well. When the beam reached the boundaries which were perfect conducts it reflected and affected the particles again. The particles gained acceleration in the x direction then they moved forward rapidly. As particles are crossing the boundaries of the cells the induced current could

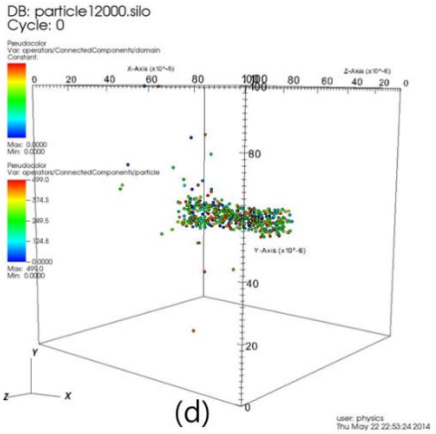
be generated along the cell's boundaries. This induced current may create new electric fields that induce the magnetic field again. In this simulation, however, we did not treat this work but only the particle movement using the CUDA method.

The simulation in the three-dimensional particle movement is not too different. Except for the z direction component, it is similar to the source code for the two-dimensional simulation. Fig. 15 shows the results of the particle movement in the three-dimensional domain.









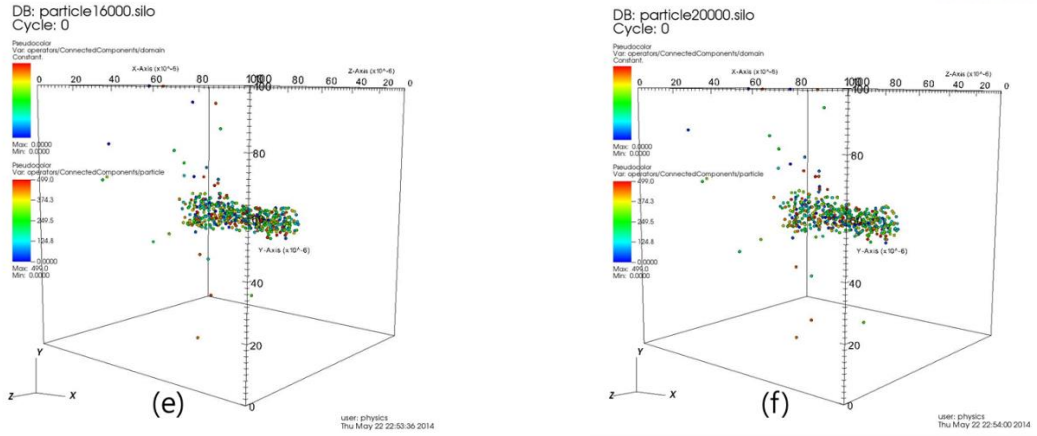


Fig. 15. Particles movement by Gaussian beam in the three-dimensional FDTD simulation

6. Future work

Up to now, the CUDA algorithm was discussed the efficiency of the FDTD simulation in the parallel computation was compared with the MPI method. Additionally, particle movement in the two and three-dimensional FDTD simulation using the CUDA method was introduced. There are few works that use the simulation to solve more difficult and complex problems. In the simulation, there was nothing but particles and the Gaussian beam in the domain. A function needs to process that some objects that exist in the domain interact with the Gaussian beam or particles. Sometimes, an object like mirror, lens, or any other device can exist in the simulation to make various experiments. The interaction between particles and device or beam and device will happen during implementing the simulation, so a function for the situation is needed.

The particle movement in the simulation presented the particles moved by the Gaussian beam that was reflected at the boundaries of the domain. This phenomenon makes it difficult to conduct an experiment on PIC (Particle-In-Cell) simulation because unwanted beams distracts all particles and makes them move in the wrong direction and makes deviation. In order to deal with the reflection, PML (Perfectly Matched Layer) theory should be introduced. PML as a virtual layer in the simulation can weaken the incident beam by distracting in the layer and reflect the beam. The PML has an effect on absorption like as if a piece of sponge absorbs water. The reflected beam from the PML has very little effect on the particles.

The efficiency of the CUDA method with hardware technique can be considered in the FDTD simulation. In this paper, only global memory on the graphic card was used for the computation. The shared memory which is an on-chip memory like a cash memory makes the computation faster by implementing the simulation. If the algorithm can be modified for using the shared memory it will decrease the computation time.

7. Conclusion

The experiment with the CUDA method and the MPI method demonstrated to compare their calculation speed. Generally, the CUDA method using the GPU's thread is better than the MPI method. Since it is difficult to speed up the CPU processor due to technological limitations, the MPI method is not available all the time. The CUDA method for the parallel computation is one of the new alternatives to solve the problem. Although graphic card has a limitation with memory size and thread number, it can be excellent solution by optimizing the simulation conditions.

REFERENCES

1. V. Demir, A. Z. Elsherbeni, Aces Journal, vol. 25, No. 4, April 2010.
2. N. Takada, N. Masuda, T. Tanaka, Y. Abe, T. Ito, Applied Computational Electromagnetics Society Journal, vol. 23, no. 4, 2008.
3. P. Sypek, A. Dziekonski, M. Mrozowski, IEEE Transactions on Magnetics, vol. 45, no. 3, 2009.
4. P. Sypek, M. Mrozowski, 17th International Conference on Microwaves, Radar and Wireless Communications (MIKON), May 2008.
5. A. Valcarce, G. De La Roche, A. Juttner, D. Lopez-Perez, J. Zhang, EURASIP Journal on Wireless Communications and Networking, Feb. 2009.
6. K. S. Yee, IEEE Transactions on Antennas and Propagation, vol. 14, May 1966.
7. D. De Donno, A. Esposito, L. Tarricone, L. Catarinucci, IEEE Antennas and Propagation Magazine, vol. 52, No. 3, June 2010
8. D. K. Price, J. R. Humphrey, E. J. Kelmelis, Physics and Simulation of Optoelectronic Devices XV, of Proc. of SPIE, vol. 6468, Jan. 2007.
9. David J. Griffiths, Introduction to Electrodynamics, 3rd edition, Prentice Hall.
10. Stephen T. Thornton, A. Rex, Modern Physics for Scientists and Engineers, 3rd edition, Thomson Books/Cole.
11. T. Marion, Classical Dynamics of Particles and Systems, 5th edition, Thomson Books/Cole.
12. NVIDIA CUDA ZONE

    http://www.nvidia.co.kr/object/cuda-kr.html
13. CUDA Education & Training

    https://developer.nvidia.com/cuda-education-training